Scalable Trigger Processing in TriggerMan

Eric N. Hanson, Mohan Konyala, Lloyd Noronha, J. B. Park and Albert Vernon
301 CSE
CISE Department
University of Florida
Gainesville, FL 32611-6120
(352) 392-2691
hanson@cise.ufl.edu
http://www.cise.ufl.edu/~hanson

TR-98-008; Version 3
9 July 1998

Abstract

Current database trigger systems have extremely limited scalability. Some commercial systems allow only one trigger per table to be defined for each type of update event. Application systems could use the ability to create thousands or even millions of triggers. The advent of the Internet and the World Wide Web has made it even more important that trigger systems scale up since users could create large number of triggers via a web interface. This paper presents a way to develop a truly scalable trigger system. Scalability is achieved with a trigger cache to use main memory effectively, and a memory-conserving selection predicate index based on the use of unique expression formats called expression signatures. A key observation is that if a very large number of triggers are created, most will have the same structure, except for the appearance of different constant values. Expression signatures are used to divide selection predicates of triggers into equivalence classes. When a trigger is created, tuples are added to special relations created for expression signatures if needed. These tables can be augmented with a database index or main-memory index structure to serve as a predicate index.

1. Introduction

Trigger features in commercial database products are quite popular with application developers since they allow integrity constraint checking, alerting, and other operations to be performed uniformly across all applications. Unfortunately, effective use of triggers is hampered by the fact that current trigger systems in commercial database products do not scale. Numerous database products only allow one trigger for each type of update event (insert, delete and update) on each table.

Application designers could effectively use large numbers of triggers (thousands or even millions) in a single database if it were feasible. The advent of the Internet and the World Wide Web makes it even more important that it be possible to support large numbers of triggers. A web interface could allow users to interactively create triggers over the Internet. This type of architecture could lead to large numbers of triggers created in a single database.

This paper presents strategies for developing a highly scalable trigger system. The concepts introduced here are being implemented in an extension module for an object-relational DBMS (a DataBlade for Informix with Universal Data Option, hereafter simply called Informix [Info98]). The system we are developing, called TriggerMan, uses asynchronous trigger processing and a sophisticated predicate index to give good response time for updates, while still allowing processing of large numbers

This work was supported in part by grants from the Defense Advanced Research Projects Agency and NCR/Teradata Corporation, and a software donation from Informix Corporation.

of potentially expensive triggers. The scalability concepts outlined in this paper could also be used in a trigger system inside a DBMS server.

A key concept that can be exploited to develop a scalable trigger system is that if a large number of triggers are created, it is almost certainly the case that many of them have almost the same format. Many triggers may have identical structure except that one constant has been substituted for another, for example. We take advantage of this by identifying unique *expression signatures*, and grouping predicates taken from trigger conditions into equivalence classes based on these signatures.

The number of distinct expression signatures is fairly small, small enough that main memory data structures can be created for all of them. In what follows, we discuss the TriggerMan command language and architecture, and then turn to a discussion of how large numbers of triggers can be handled effectively using expression signature equivalence classes and a novel selection predicate indexing technique.

2. The TriggerMan Command Language

Commands in TriggerMan have a keyword-delimited, SQL-like syntax. TriggerMan supports the notion of a *connection* to a local Informix database, a remote database, or a generic data source program. A connection description for a database contains information about the host name where the database resides, the type of database system running (e.g. Informix, Oracle, Sybase, DB2 etc.), the name of the database server, a user ID, and a password. A single connection is designated as the default connection. There can be multiple *data sources* defined for a single connection. Data sources normally correspond to tables, but this is not essential. Data sources can be defined using this command:

```
define data source [connectionName.]sourceName [as localName]
[ ( attributeList ) ]
[ propertyName=propertyString,
...
  propertyName=propertyString ]
```

Suppose a connection “salesDB” had been defined on a local database called “sales.” An example data source definition for the table “sale” in the sales database might look like this:

```
define data source salesDB.sale as sale
```

This command would read the schema from the salesDB connection for the “sale” table to gather the necessary information to allow triggers to be defined on that table.

Triggers can be defined using the command shown at right.

```
create trigger <triggerName> [in setName]
[optionalFlags]
from fromList
[on eventSpec]
[start timePoint]
[end timePoint]
[calendar calendarName]
[when condition]
[group by attr-list]
[having group-condition]
do action
```

The **start** and **end** clauses define a time interval within which the trigger can be eligible to run. In addition, the name of a calendar object can be specified as part of a trigger. A calendar indicates “on” and “off” time periods. For example, a simple business calendar might specify “on” periods to be Monday through Friday from 8:00AM to 5:00PM. A trigger with a calendar is only eligible to be triggered during an “on” period for the calendar. In addition, a trigger is eligible to be triggered only if:

(1) the trigger is active, and

(2) the current time is between the start time and end time, and

(3) the calendar is in an "on" time period.

Triggers can be added to a specific trigger set. Otherwise they belong to a default trigger set. The **from**, **on**, and **when** clauses are normally present to specify the trigger condition. Optionally, **group by** and **having** clauses, similar to those available in SQL [Date93], can be used to specify trigger conditions involving aggregates or temporal functions. Multiple date sources can be referenced in the **from** clause. This allows multiple-table triggers to be defined.

An example of a rule, based on an **emp** table from a database for which a connection has been defined, is given below. This rule sets the salary of Fred to the salary of Bob:

```
create trigger updateFred
from emp
on update emp.salary
when emp.name = "Bob"
do execSQL "update emp set salary=:NEW.emp.salary where emp.name= 'Fred'"
```

This rule illustrates the use of an execSQL TriggerMan command that allows SQL statements to be run against a database. The :NEW notation in the rule action (the **do** clause) allows reference to new updated data values, the new emp.salary value in this case. Similarly, :OLD allows access to data values that were current just before an update. Values matching the trigger condition are substituted into the trigger action using macro substitution. After substitution, the trigger action is evaluated. This procedure binds the rule condition to the rule action.

An example of a more sophisticated rule (one whose condition involves joins) is as follows. Consider the following schema for part of a real-estate database, which would be imported by TriggerMan using **define data source** commands:

```
house(hno,address,price,nno,spno)
salesperson(spno,name,phone)
represents(spno,nno)
neighborhood(nno,name,location)
```

A rule on this schema might be "if a new house is added which is in a neighborhood that salesperson Iris represents then notify her," i.e.:

```
create trigger IrisHouseAlert
on insert to house
from salesperson s, house h, represents r
when s.name = 'Iris' and s.spno=r.spno and r.nno=h.nno
do raise event NewHouseInIrisNeighborhood(h.hno, h.address)
```

This command refers to three tables. The **raise event** command used in the rule action is a special command that allows rule actions to communicate with the outside world [Hans98]. Application programs written using a library provided with TriggerMan can register for events. When triggers raise events, the applications registered for the events will be notified. Applications can run on machines running anywhere on the network that is reachable from the machine where TriggerMan is running.

3. System Architecture

The TriggerMan architecture is made up of the following components:

1. the **TriggerMan DataBlade** which lives inside of Informix,
2. **data source applications**, which are programs that transmit a sequence of *update descriptors* to TriggerMan (update descriptors describe updates that have occurred in data sources, including the operation that was performed, and a new tuple, old tuple, or old/new pair in the case of insert, delete and update operations, respectively),
3. **TriggerMan client applications**, which create triggers, drop triggers, register for events, receive event notifications when triggers fire, etc.,
4. one or more instances of the **TriggerMan driver** program, each of which periodically invokes a special TmanTest() function in the TriggerMan DataBlade, allowing trigger condition testing and action execution to be performed,
5. the **TriggerMan console**, a special application program that lets a user directly interact with the system to create triggers, drop triggers, start the system, shut it down, etc.

The general architecture of the TriggerMan system is illustrated in Figure 1. Two libraries that come with TriggerMan allow writing of client applications and data source programs. These libraries define the TriggerMan *client application programming interface* (API) and the TriggerMan *data source API.* The console program and other application programs use client API functions to connect to TriggerMan, issue commands, register for events, and so forth. Data source programs can be written using the data source API. Examples of data source programs are a generic data source that sends a stream of update descriptors to TriggerMan, and a DBMS gateway program that gathers updates from a DBMS and sends them to TriggerMan. Updates to data source tables can also be gathered by placing simple Informix insert, update and delete triggers on the tables. These triggers place "update descriptors" into tables used to queue updates for TriggerMan. These updates are consumed on the next call to TmanTest().

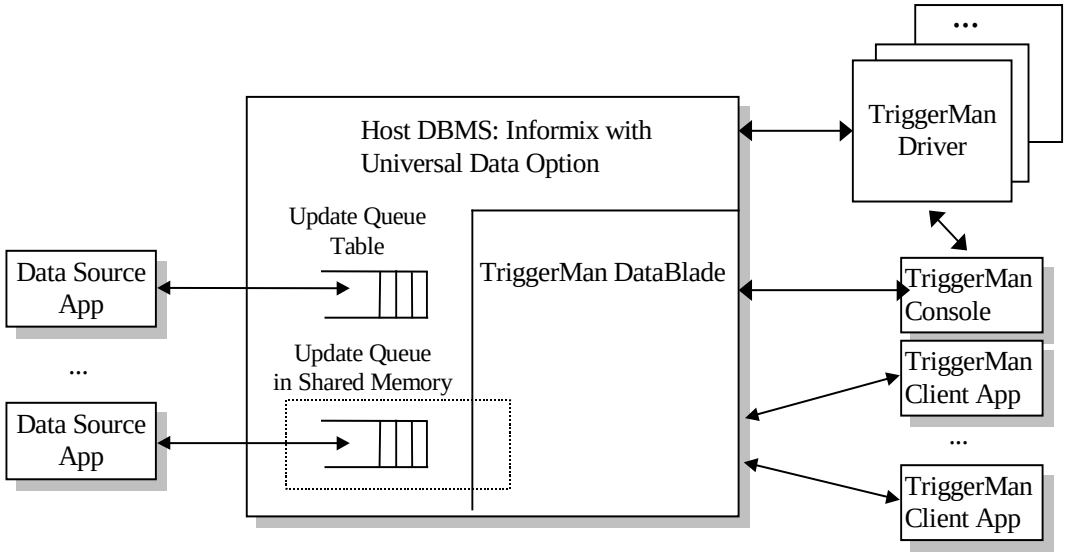


Figure 1. The architecture of the TriggerMan trigger processor.

As Figure 1 shows, data source programs or triggers can either place update descriptors in a persistent table acting as a queue, or in a volatile queue in shared memory. If simple Informix triggers are used to capture updates, TriggerMan can process its own triggers either synchronously (as part of the current update transaction) or asynchronously (in a separate transaction). A volatile queue in shared memory is used to hold update descriptors that do not need to be recovered in case of system failure. It is up to the application designer to determine if update descriptors sent to TriggerMan need to be recoverable. The volatile queue can actually be implemented as a main memory database table (a feature provided by Informix) to simplify the software design.

TriggerMan is based on an object-relational data model. Attributes of tuples and update descriptors manipulated by TriggerMan can be either simple data types, or user-defined types. These user-defined types correspond to user-defined types in TriggerMan data sources. User-defined routines (functions, methods, or operators) can be used in TriggerMan conditions. The design of TriggerMan's trigger condition testing strategy takes into account the cost of execution of user-defined routines.

4. Overview of TriggerMan Trigger Processing Strategy

The idea behind the trigger processing strategy used in TriggerMan is as follows. TriggerMan uses a discrimination network called a *Gator network* [Hans97b] for trigger condition testing. A Gator network tree structure will be constructed for each trigger. Gator networks consist of nodes to test selection and join conditions, plus "memory" nodes that hold sets of tuples matching one or more selection and join conditions. Memory nodes that hold the result of applying a selection condition are called alpha nodes. Memory nodes that hold the result of the join of two or more other memory nodes are called beta nodes. Beta nodes can have two or more other memory nodes as inputs. In other words, the Gator network tree need not be a binary tree. It can be a bushy rooted tree, where the root is normally drawn at the bottom. In Rete network terminology, the root is called the P-node. As an example, consider the following table schemas and trigger definition:

```
R1(r1no,a,b)
R2(r1no,r3no)
R3(r3no,c,d)

create trigger T1
from R1, R2, R3
when R1.r1no=R2.r1no
and R2.r3no=R3.r3no
and R1.a = "x"
then do ...
```

The Gator network for this trigger is shown in Figure 2. In this Gator network, memory alpha1 logically contains the result of

```
select * from R1 where R1.a= "x"
```

Similarly, beta1 logically contains the result of

```
select * from R1, R2 where R1.a = "x" and R1.r1no=R2.r1no
```

In addition, memory nodes can be either stored or virtual. A stored node is analogous to a materialized view. It actually contains the specified tuples. A virtual node is analogous to a real view. It only contains a predicate defining which tuples should qualify. It does not contain the real tuples. Only alpha memory nodes can be virtual.

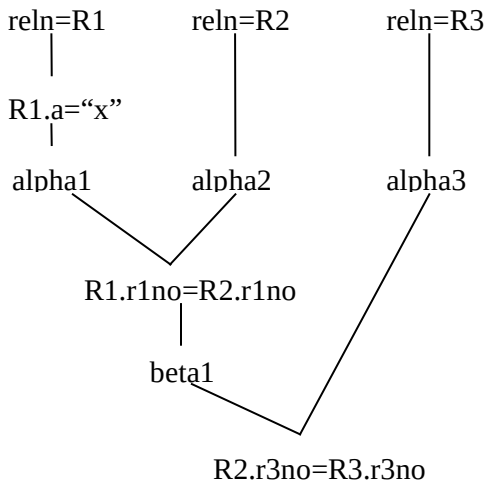


Figure 2. Gator network for trigger T1.

A detailed discussion of how to choose the best shape of a Gator network, and how Gator networks can be used for multiple-table trigger condition testing on a single processor can be found in a separate paper [Hans97b]. A presentation of how the Gator network optimizer for TriggerMan will work is beyond the scope of this paper. However, techniques based on the work shown in our prior paper [Hans97b] as well as work by Kandil [Kand98] will be used. A discussion of how updated data tuples (also called update descriptors or tokens) are processed using Gator networks to see when a particular trigger should fire is included in [Hans97b].

5. General Trigger Condition Structure

Trigger conditions have the following general structure. The **from** clause refers to one or more data sources. The **on** clause may contain an event condition for at most one of the data sources referred to in the **from** list. The **when** clause of a trigger is a Boolean-valued expression. For a combination of one or more tuples from data sources in the **from** list, the **when** clause evaluates to true or false.

A canonical representation of the **when** clause can be formed in the following way:

1. Translate it to conjunctive normal form (CNF, or and-of-ors notation).
2. Each conjunct refers to zero, one, two, or possibly more data sources. Group the conjuncts by the set of data sources they refer to.

If a group of conjuncts refers to one data source, the logical AND of these conjuncts is a selection predicate. If it refers to two data sources, the AND of its conjuncts is a join predicate. If it refers to zero conjuncts, it is a trivial predicate. If it refers to three or more data sources, we call it a hyper-join predicate.

These predicates may or may not contain constants. The general premise of this paper is that very large numbers of triggers will only be created if predicates in different triggers contain distinct constant values. Below, we will examine how to handle selection and join predicates that contain constants, so that scalability to large numbers of triggers can be achieved.

6. Scalable Predicate Indexing Using Expression Signatures

In what follows, we treat the event (**on**) condition separately from the **when** condition as a convenience. However, event conditions and **when** clause conditions are both logically selection conditions [Hans96] that can be applied to update descriptors submitted to the system.

A *tuple variable* is a symbol, defined in the **from** clause of a trigger, which corresponds to a usage of a particular data source in that trigger. The general form of a selection predicate is:

$$(C_{11} \text{ OR } C_{22} \text{ OR } ... \text{ OR } C_{1N_1}) \text{ AND } ... \text{ AND } (C_{K1} \text{ OR } C_{K2} \text{ OR } ... \text{ OR } C_{KN_K})$$

where all clauses C_{ij} appearing in the predicate refer to the *same* tuple variable. Furthermore, each such clause is an atomic expression that does not contain Boolean operators, other than possibly the NOT operator. A single clause may contain constants.

The general form of a join predicate is the same, except that the clauses C_{ij} can refer to two tuple variables. All the clauses C_{ij} in a single join predicate must refer to the *same* pair of tuple variables.

The general form of trivial and hyper-join predicates can also be defined in a comparable fashion. We will not elaborate on the details of those here since they are rare special cases.

For convenience, we assume that every data source has a data source ID. A data source can be either a simple data source or a compound data source. A simple data source corresponds to a single table in a remote or local database, or even a single stream of tuples sent in messages from an application program. A compound data source generates update descriptors consisting of joining combinations of tuples from two or more simple data sources. An expression signature for a general selection or join predicate expression is a triple consisting of a data source ID, an operation code (insert, delete, update, or insertOrUpdate), and a generalized expression. If a tuple variable appearing in the **from** clause of a trigger does not have any event specified in the **on** clause, then the event is implicitly insert *or* update for that tuple variable. The format of the generalized expression is:

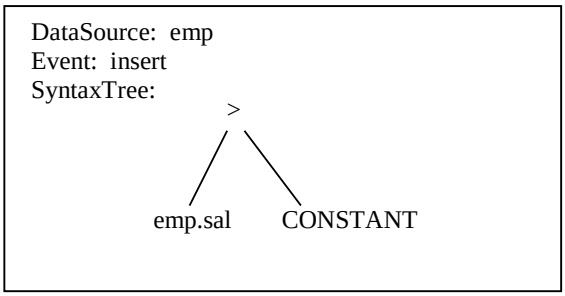
$$(C'_{11} \text{ OR } C'_{22} \text{ OR } ... \text{ OR } C'_{1N_1}) \text{ AND } ... \text{ AND } (C'_{K1} \text{ OR } C'_{K2} \text{ OR } ... \text{ OR } C'_{KN_K})$$

where clause C'_{ij} is the same as C_{ij} except that all constants in C_{ij} are substituted with placeholder symbols. If the entire expression has m constants, they are numbered 1 to m from left to right. If the constant number $x, 1 \le x \le m,$ appears in the clause C_{ij} in the original expression, then it is substituted with placeholder CONSTANT_X in C_{ij} in the expression signature.

As a practical matter, most selection predicates will not contain OR's, and most will have only a single clause. Consider this example trigger condition:

```
on insert to emp
when emp.salary > 80000
```

In an implementation, the generalized expression in an expression signature can be a *syntax tree* with placeholders at some leaf nodes representing the location where a constant must appear. For example, the signature of the trigger condition just given can be represented as:



The condition

on insert to emp
when emp.salary > 50000

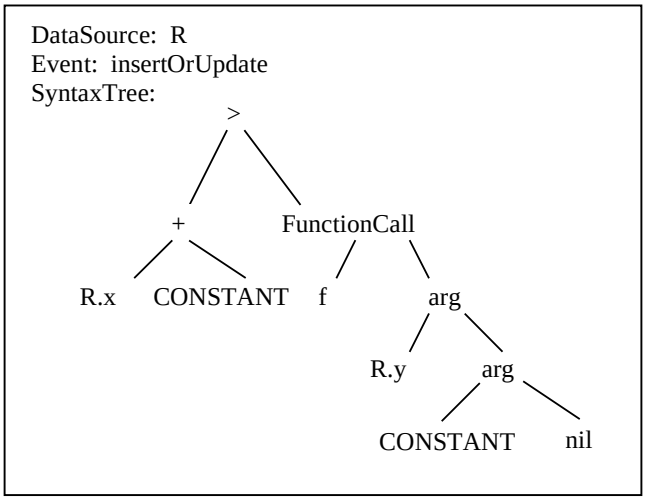
has a different constant than the earlier condition, but it has the same signature. In general, an expression signature defines an *equivalence class* of all instantiations of that expression with different constant values. Hence, the two expressions in the example just given are in the same equivalence class.

If an expression is in the equivalence class defined by an expression signature, we say the expression *matches* the expression signature. A test for this match can be implemented using a recursive function that traverses the syntax trees for the expression and the expression signature in tandem.

Any kind of expression, even one involving function calls, can be reduced to a single expression signature. For example, consider a trigger condition with no reference to data source R in the **on** clause, and this **when** expression:

R.x + 100 > f(R.y, 200)

This trigger condition has the expression signature shown in the following figure.



Expression signatures represent the logical structure or schema of a part of a trigger condition. We assert that in a real application of a trigger system like TriggerMan, even if very large numbers of triggers are defined, only a relatively small number of unique expression signatures will ever be observed - perhaps a few hundred or a few thousand at most. Based on this observation, it is feasible to keep a set of data structures in main memory to represent all the distinct expression signatures appearing in all triggers. Since many triggers may have the same signature but contain different constants, tables will be

created to store these constants, along with information linking them to their expression signature. When these tables are small, low-overhead main-memory lists or indexes can be used to cache information from them. When they are large, the can be stored as standard tables (perhaps with an index) and queried as needed using the SQL query processing to perform trigger condition testing. We will elaborate further on implementation issues below.

6.1. Processing a Trigger Definition

When a **create trigger** statement is processed, a number of steps must be perform to update the trigger system catalogs and main memory data structures, and to "prime" the trigger to make it ready to run. The primary tables that form the trigger catalogs are these:

trigger_set(tsID, name, comments, creation_date, isEnabled)
trigger(triggerID, tsID, name, comments, trigger_text, creation_date, isEnabled, …)

The purpose of the isEnabled field is to indicate whether a trigger or trigger set is currently enabled and eligible to fire if matched by some update. The other fields are self-explanatory. A data structure called the *trigger cache* is maintained in main memory. This contains complete descriptions of a set of recently accessed triggers, including the trigger ID and name, references to data sources relevant to the trigger, and the syntax tree and Gator network skeleton for the trigger. Given current main memory sizes, thousands of trigger descriptions can be loaded in the trigger cache simultaneously. E.g. if a trigger description takes 4K bytes (a realistic number), and 64Mbytes are allocated to the trigger cache, 16,384 trigger descriptions can be loaded simultaneously.

Another main memory data structure called a *predicate index* is maintained. A diagram of the predicate index is shown in Figure 3. The predicate index can take an update descriptor and identify all predicates that match it.

Expression signatures may contain more than one conjunct. If a predicate has more than one conjunct, a single conjunct is identified as the most selective one. Only this one is indexed directly. If a token matches a conjunct, any remaining conjuncts of the predicate are located and tested against the token. If the remaining clauses match, then the token has completely matched the predicate clause. See [Hans90] for more details on this technique.

The root of the predicate index is linked to a set of *data source predicate indexes* organized using a hash table on data source ID. Each data source predicate index contains an *expression signature list* with one entry for each unique expression signature that has been used by one or more triggers as a predicate on that data source. For each expression signature that contains one or more constant placeholders, there will be a *constant table.* This is an ordinary database table containing one row for each expression occurring in some trigger that matches the expression signature.

When triggers are created, any new expression signatures detected are added to the following table in the TriggerMan system catalogs:

expression_signature(sigID,dataSrcID, signatureDesc, constTableName,
constantSetSize, constantSetOrganization)

The sigID field is a unique ID for a signature. The dataSrcID field identifies the data source on which the signature is defined. The signatureDesc field is a text field with a description of the signature. We will define the other fields later.

When an expression signature E is encountered at trigger creation time, it is broken into two parts: the indexable part, E_I, and the non-indexable part, E_NI, as follows:

E = E_I AND E_NI

The non-indexable portion may be NULL. The format of the constant table for an expression signature containing K distinct constants in its indexable portion is:

const_tableN(exprID,triggerID,nextGatorNetNode,const1, … constK,restOfPredicate)

Here, N is the identification number of the expression signature. The fields of const_tableN have the following meaning:

1. exprID is the unique ID of a selection predicate E,
2. triggerID is the unique ID number of the trigger containing E,
3. nextGatorNetNode identifies the next Gator network node of trigger triggerID to pass a token to after it matches E,
4. const1 … constK are constants found in the indexable portion of E, and
5. restOfPredicate is a description of the non-indexable part of E. The value of restOfPredicate is NULL if the entire predicate is indexable.

If the table is large, and the signature of the indexable part of the predicate is of the form attribute1=CONSTANT1 AND … attributeK=CONSTANTK, the table will have a clustered index on [const1, … constK] as a composite key. If the predicate has a different type of signature based on an operator other than "=", it may still be possible to use an index on the constant fields. As future work, we propose to develop ways to index for non-equality operators and constants whose types are user-defined [Kony98].

Putting a clustered index on the constant attributes will allow the triggerIDs of triggers relevant to a new update descriptor matching a particular set of constant values to be retrieved together quickly without doing random I/O. Notice that const_tableN is not in third normal form. This was done purposely to eliminate the need to perform joins when querying the information represented in the table.

Referring back to the definition of the expression_signature table, we can now define the remaining attributes:

1. constTableName is a string giving the name of the constant table for an expression signature,
2. constantSetSize is the number of distinct constants appearing in expressions with a given signature, and
3. constantSetOrganization describes how the set of constants will be organized in either a main-memory or disk-based structure to allow efficient trigger condition testing. The issue of constant set organization will be covered more fully later in the paper.

Given the disk- and memory-based data structures just described, the steps to process a **create**

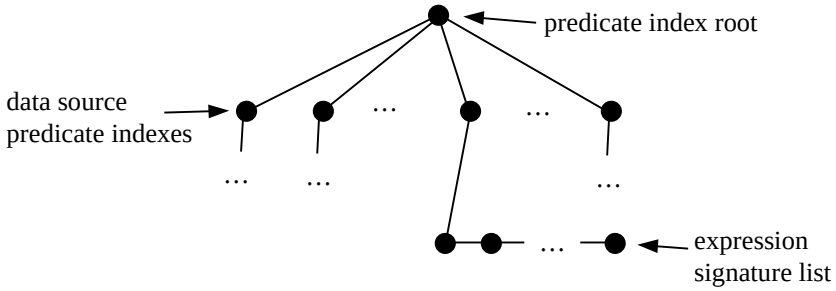


Figure 3. Predicate Index Structure.

trigger statement are:

1. Parse the trigger and validate it (check that it is a legal statement).
2. Convert the **when** clause to conjunctive normal form and group the conjuncts by the distinct sets of tuple variables they refer to, as described in section 5.
3. Based on the analysis in the previous step, form a trigger condition graph. This is an undirected graph with a node for each tuple variable, and an edge for each join predicate identified. The nodes contain a reference to the selection predicate for that node, represented as a CNF expression. The edges each contain a reference to a CNF expression for the join condition associated with that edge. Groups of conjuncts that refer to zero tuple variables or three or more tuple variables are attached to a special "catch all" list associated with the query graph. These will be handled as special cases. Fortunately, they will rarely occur. We will ignore them here to simplify the discussion.
4. Run the Gator network optimizer for the rule on the rule condition graph, forming an optimized Gator network. Most of the selection predicates will be at the leaves, but some expensive selections may appear lower in the network, after a beta node [Kand98,Hans97b].
5. For each selection predicate at the leaf level in the Gator network, do:

   Check to see if its signature has been seen before by comparing its signature to the signatures in the expression signature list for the data source on which the predicate is defined (see Figure 3). If no predicate with the same signature has been seen before,

   - add the signature of the predicate to the list and update the expresssion_signature catalog table.
   - If the signature has at least one constant placeholder in it, create a constant table for the expression signature.

   If the predicate has one or more constants in it, add one row to the constant table for the expression signature of the predicate.

This concludes the basic description of how to process a **create trigger** command. However, to this point we have only dealt with expressions containing constants that appear as leaf-level selection conditions in a Gator network. Constants can also appear in selection predicates that have been pushed down below joins [Kand98], and in join predicates. It is possible that a large number of rules with the same expression signature for pushed-down selections or joins, but different constants, could be created. These cases would be expected to be rare in real applications. Moreover, handling them efficiently requires an additional degree of complexity. Hence, we propose the following alternative implementation levels to handle pushed-down selections and joins containing constants:

Level 1: if selections are pushed down, do not store signatures for them. When one of the leaf-level selection conditions of the Gator network for the trigger containing the pushed-down selection is matched by an update descriptor, bring the complete trigger definition and Gator network into the trigger cache if it is not there already. The Gator network may contain constants in the pushed-down selection predicates. If there are not many distinct constants in pushed-down selection predicates for the entire set of triggers, this will work well. It is likely that this will be the case. In the unlikely case that there is a pushed down selection predicate signature common to a large number of triggers with different constants, this level 1 approach could cause the trigger cache to be flushed frequently, hurting performance. However, it will still be possible to handle large numbers of triggers of this type -- the system will not run out of main memory.

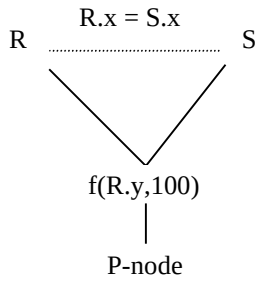
Level 2: if selections are pushed down, store signatures for them only if no constants appear in the Gator network above the selection. The join of the relations above the selection is a compound data

source. Since no constants can appear in the definition of one of these compound data sources, the number of them will not be too large. Hence, descriptions of all of them can be kept in main memory as part of the predicate index shown in Figure 3.

An example of a rule condition that involves an expensive function f that references a constant is:

from R, S
on insert to R
when R.x = S.x and f(R.y,100)

The Gator network optimizer might decide to build the following Gator network for this rule condition:



This is an example of a pushed-down select. Using the Level 2 approach, the join of R and S on x would be identified as a compound data source, and there would be a data source predicate index for it. One of the entries in the expression signature list for this data source would be f(R.y,CONSTANT). One of the constants recorded for this signature would be 100. Constants for additional triggers with the same format would also be recorded for this signature.

6.2. Optimized Handling of OR Predicates

A potential topic for future work is to optimize the processing of selection predicates containing ORs. In general, when a conjunct of a selection predicate contains ORs, none, some or all of the ORed clauses may be true when it is applied to a token. Hence, if all of the clauses are indexed, multiple clauses may in general match a newly arriving token. Some duplicate match elimination must thus be used. This is analogous to the need to do a UNION of the results of several independent subqueries if an SQL **select** statement with OR's in its **where** clause is processed using independent subqueries. When a sequential scan is used to process a selection involving OR's, no UNION is necessary.

Our initial approach to handling OR conditions is to treat them as non-indexable. Hence, every token from the data source on which the OR condition is defined would be tested against the entire OR condition. As an example of a trigger condition with OR's, consider this:

on insert to stockTick
when symbol = "IBM" OR symbol = "HP" OR symbol = "SUNW"

The approach initially used will be to test every insert token to stockTick against the condition:

symbol = "IBM" OR symbol = "HP" OR symbol = "SUNW"

However, this approach is overly conservative in this example. In general, if a predicate is of the form (C_1OR C_2 OR … C_N) and each clause C_i is of the form attribute=CONSTANT for the same attribute, and each CONSTANT only appears once, than at most one clause can be true for a newly arriving token. This eliminates the need to do duplicate match elimination. Each one of the clauses could be indexed separately. This could give a tremendous speedup if a large number of triggers with conditions of this

type were created. We propose to design extensions to our predicate index structure to take advantage of this optimization.

6.3. Alternative Organization Strategies for Expression Equivalence Classes

For a particular expression signature that contains at least one constant placeholder, there may be one or more expressions in its equivalence class that belong to different triggers. This number could be small or large. To get optimal performance over a wide range of sizes of the equivalence classes of expressions for a particular expression signature, alternative indexing strategies are needed. Main-memory data structures with low overhead are needed when the size of an equivalence class is small. Disk-based structures, including indexed or non-indexed tables, are needed when the size of an equivalence class is large.

We consider the following four ways to organize the predicates in an expression signature's equivalence class:

1. main memory list
2. main memory index
3. non-indexed database table
4. indexed database table

Strategies 3 and 4 *must* be implemented to make it feasible to process very large numbers of triggers containing predicate expressions with the same signature but different constants -- they are mandatory in a scalable trigger system.

An example of how 3 and 4 would work is as follows. Suppose there are 1,000,000 triggers of the form:

```
create trigger t_I
from R
on insert to R
when R.x = CONSTANT_I
```

Here, CONSTANT_I represents a unique constant for trigger t_I. All these trigger's have a **when** clause containing a selection predicate with the same signature. Suppose this signature was assigned number N. A constant table for this signature would be created with the following schema:

```
const_table_N(exprID,triggerID,nextGatorNetNode,const,restOfPredicate)
```

When a token t describing an insert to R arrives, the following SQL statement would be formed:

```
select * from const_table_N where const = t.x
```

Here, t.x is actually a constant extracted from t, and const is an attribute of const_table_N. The query processor then executes this statement. If there is no index on the const_table_N.const attribute, this is strategy 3. If there is an index on that attribute, this is strategy 4. For each tuple retrieved by the above query, the appropriate trigger would be located and executed.

Organizations 1 and 2 are optional. However, they represent the common case, so implementing them can pay off handsomely. To show the variation in cost between the four schemes for relatively small sets of predicates that have a particular signature, a cost analysis is given below. Since the sets are small, it is assumed that the table and index structures fit in main memory for organizations 3 and 4, so I/O cost is ignored.

In 1, a main-memory list of predicates, including any constants they contain, is maintained. When a token is tested to see which predicates match, the list is scanned sequentially and each predicate is tested against the token. Let N be the size of the set of predicates. The cost of 1 can be modeled as:

COST_MM_LIST = C1 + C2*N

The overhead is C1 and the per-predicate cost is C2. For 2, an appropriate type of index would be maintained depending on the structure of predicate. Assuming the predicate type is attribute=CONSTANT and the index is a balanced binary search tree, the cost can be modeled as:

COST_MM_INDEX = C3 + C4* $\log_2 N$

For 3, the cost can be modeled as:

COST_TBL_NO_INDEX = C5 + C6*N

For 4, assuming an equality predicate and a B+-tree index on the table, the cost can be modeled as:

COST_TBL_INDEX = C7 + C8* $\lceil \log_B N \rceil$

where B is the B+-tree index fanout.

We performed measurements using Informix and Microsoft Visual C on a 333Mhz Pentium II computer to obtain C5 and C6, and estimated C1-C4 and C7 and C8. These values are included in the table below, along with an estimate for B:

variable	**value**
C1	100 μs
C2	50 μs
C3	200 μs
C4	75 μs
C5	1000 μs
C6	600 μs
C7	3000 μs
C8	1000 μs
B	50

Graphs of the cost of the different approaches are shown in Figure 4 for sets of size 1 to 250 predicates. It takes about 0.16 seconds per token in the worst case, which is the table/no-index strategy for 250 predicates, shown in Figure 4 (A). This would allow only 6 tokens per second to be processed. Since the table/no-index strategy performs significantly worse than the others, Figure 4 (B) is given to show the cost of the others on a less compressed scale. Notice that the cost of the main-memory list strategy crosses over the database table/index strategy when the number of predicates is about 100. Also, in Figure 4 (B), the worst-case cost is .012 seconds to test a token against 250 predicates using the main-memory list strategy. This would allow 83 tokens to be processed per second. This only accounts for selection predicate testing costs, not other costs associated with trigger execution.

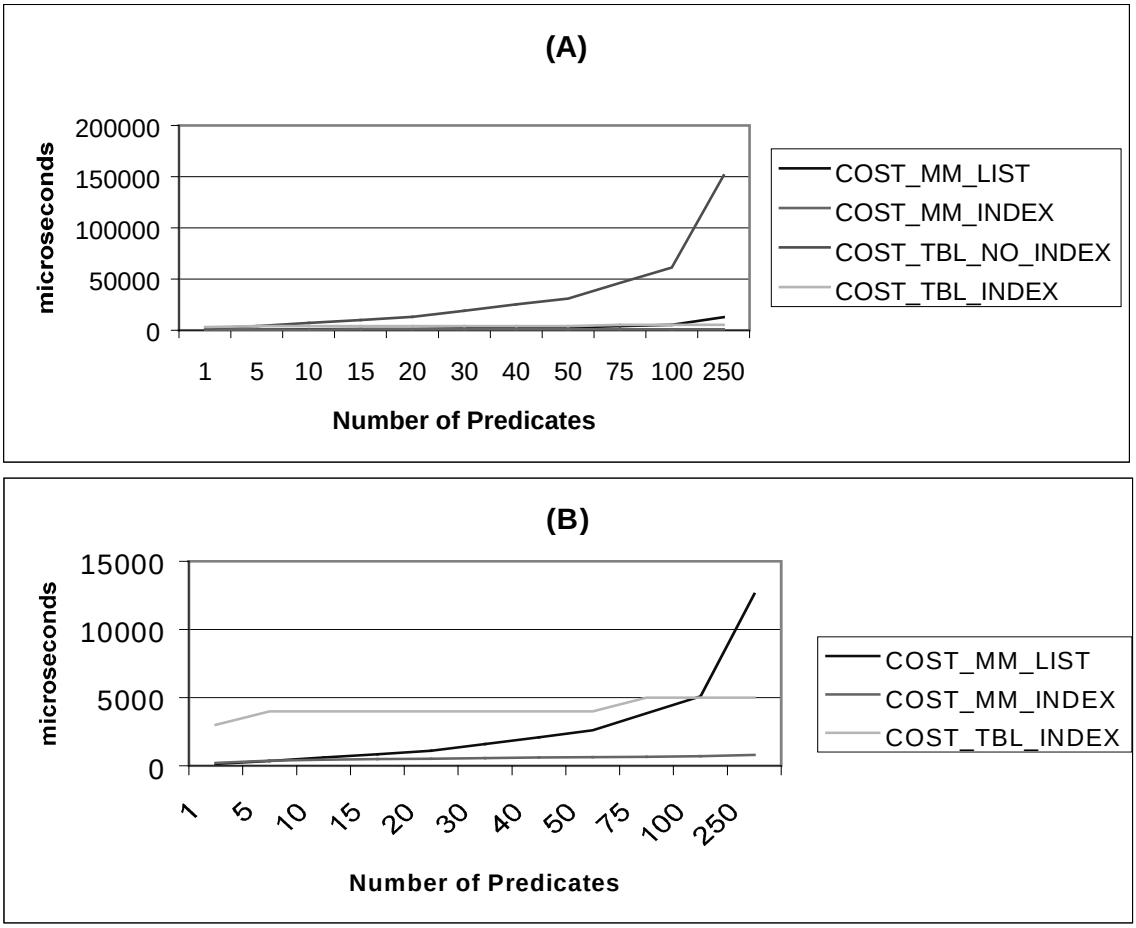


Figure 4. Cost of predicate evaluation for different predicate set organizations.

An important design goal for a scalable trigger system is that it be able to handle one thousand or more new update descriptors per second even if a large number of triggers have been defined. For example, it should be possible to take a feed of all the data updated in a large-scale OLTP system and process it in real time. The table/no-index and main-memory list strategies alone will not let us achieve this goal. Indexing strategies are crucial. Clearly, if the triggers have conditions that are very expensive to check, and it is not possible to index the conditions, then it may not be possible to keep up with a high-update-rate data source. However, for simple triggers, e.g. those with conditions of the form attribute=constant, it should be possible to handle high update rates.

Based on the analysis given in this section, we propose to implement all four of the strategies discussed, and a strategy selector that will choose the best one. Selecting the best strategy for an expression signature depends on:

1. the number of predicates in the signature's equivalence class,
2. the amount of main memory available,
3. whether the predicate is indexable or not,
4. whether the SQL query processor will run queries in parallel, and
5. the constant factor overheads in the performance curves for the different strategies.

Developing cost models and decision criteria to choose the best strategy is a subject for future research.

6.4. Common Sub-expression Elimination for Selection Predicates

An important performance enhancement to reduce the total time needed to determine which selection predicates match a token is common sub-expression elimination. This can be achieved by normalizing the predicate index structure. Figure 5 shows an expanded view of the predicate index given in Figure 3. The *constant set* of an expression signature contains one element for each constant (or tuple of constants [const1, … ,constK]) occurring in some selection predicate that matches the signature. Each constant is linked to a triggerID set, which is a set of the ID numbers of triggers containing a particular selection predicate. For example, if there are rules of the form:

create trigger T_I from R when R.a = 100 do …

for I=1 to N, then there will be an expression signature R.a=CONSTANT, the constant set for this signature will contain an entry 100, and the triggerID set for 100 will contain the ID numbers of T_1 … T_N.

We will implement constant sets and triggerID sets in a fully normalized form, as shown in Figure 5, when these sets are stored as either main memory lists or indexes (organizations 1 and 2). This normalized main-memory data structure will be built using the data retrieved from the constant table for the expression signature.

6.5. Join Predicates Involving Constants

Join predicates may also involve constants. An example of a trigger condition containing a join predicate that involves a constant is:

```
from R, S
on insert to R
when R.y > 1.1 * S.y
```

It is theoretically possible that a large number of triggers with the same format, except for different constants in a join predicate, could be created. Join conditions involving constants are rare in real applications. We feel that an implementation tuned to handle this rare special case would be complex, and the payoff would be limited to a small fraction of all applications. Hence, we will use the trigger cache mechanism to deal with triggers of this type, and not implement special predicate indexes for them.

If the prediction that trigger conditions of this type will be rare proves false, it would be possible to handle them by transforming them to pushed-down selections, and then using the Level 2 approach discussed earlier. For brevity, a detailed discussion of this concept is omitted here.

6.6. Boot-time processing

When TriggerMan boots, it reads its system catalogs and builds main-memory data structures that must always be in memory. Structures that must be in memory include the predicate index shown in Figure 5, and the set of data source schema definitions. These are constructed by reading the data source catalog table (not defined in this paper) and the expression_signature table. Trigger and trigger set definitions, and constant sets for a particular expression signature that are organized as main memory lists or index structures, are brought into memory on demand. This makes it possible to handle trigger definition and token arrival efficiently, while also keeping boot time relatively short.

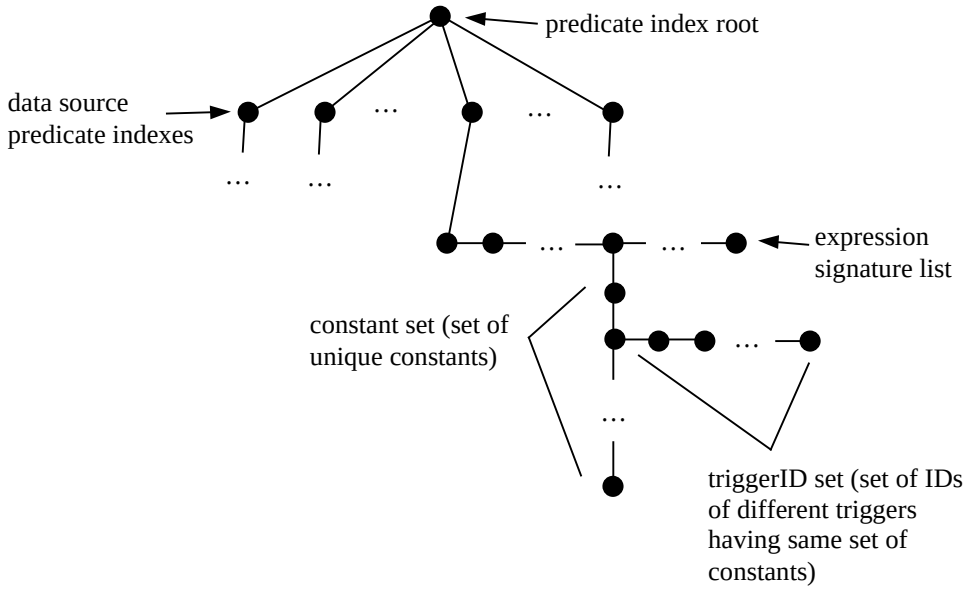


Figure 5. Expanded View of Normalized Predicate Index Structure.

6.7. Processing Update Descriptors Using the Predicate Index

With the infrastructure for testing trigger conditions in place, the process of handling newly arrived tokens can be explained. Recall that a token consists of a data source ID, an operation code, and an old tuple, new tuple, or old/new tuple pair. When a new token arrives, the system passes it to the root of the predicate index, which locates its data source predicate index. For each expression signature in the data source predicate index, a specific type of predicate testing data structure (in-memory list, in-memory lightweight index, non-indexed database table, or indexed database table) is in use for that expression signature. The predicate testing data structure of each of these expression signatures is searched to find matches against the current token.

When a matching constant is found, the triggerID set for the constant contains one or more elements. Each of these elements contains zero or more additional selection predicate clauses. For each element of the triggerID set currently being visited, the additional predicate clause(s) are tested against the token, if there are any.

When a token is found to have matched a complete selection predicate expression that belongs to a trigger, that trigger is *pinned* in the trigger cache. This pin operation is analogous to the pin operation in a traditional buffer pool; it checks to see if the trigger is in memory, and if it is not, it brings it in from the disk-based trigger catalog. The pin operation ensures that the Gator network and the syntax tree of the trigger are in main-memory. After the trigger is pinned, ensuring that its Gator network is in main memory, the token is passed to the node of the network identified by the nextGatorNetNode field of the expression that just matched the token.

Processing of join and temporal conditions is then performed if any are present. Finally, if the trigger condition is satisfied, the trigger action is executed. The syntax tree of the trigger action, which must be in memory since the trigger was pinned, is interpreted to perform this. Details of join and temporal condition processing are the subject of another paper [Hans97].

7. Concurrent Token Processing and Action Execution

An important way to get better scalability is to use concurrent processing. On an SMP platform, concurrent tasks can execute in parallel. Even on a single processor, use of concurrency can give better throughput and response time by making scarce CPU and I/O resources available to multiple tasks so any eligible task can use them. There are a number of different kinds of concurrency that TriggerMan can exploit for improved scalability:

1. **Token-level concurrency:** multiple tokens can be processed in parallel through the selection predicate index and the join condition-testing network.
2. **Condition-level concurrency:** multiple selection conditions can be tested against a single token concurrently.
3. **Rule action concurrency:** multiple rule actions that have been fired can be processed at the same time.
4. **Data-level concurrency:** a set of data values in an alpha or beta node of a Gator network [Hans97] can be processed by a query that can run in parallel.

For ideal scalability, TriggerMan must be able to capitalize on all four of these types of concurrency. TriggerMan will make use of a task queue kept in shared memory to store incoming or internally generated work. An explicit task queue must be maintained because it is not possible to spawn native operating system threads or processes to carry out tasks due to the process architecture of Informix [Info98]. If threads were available, it would be preferable to spawn multiple threads to execute multiple tasks rather than explicitly manage a task queue.

The concurrent processing architecture, as illustrated in Figure 1, will make use of N driver processes. We define NUM_CPUS to be the number of real CPUs in the system, and TMAN_CONCURRECY_LEVEL to be the fraction of CPUs to devote to concurrent processing in TriggerMan, which can be in the range (0%,100%]. The TriggerMan administrator can set the TMAN_CONCURRENCY_LEVEL parameter. Its default value is 100%. N is defined as follows:

$$N = \lceil NUM\_CPUS * TMAN\_CONCURRENCY\_LEVEL \rceil$$

Each driver process will call TriggerMan's TmanTest() function every T time units. The default value of T will be 1/4 of a second. This value of T will allow timely trigger execution without excessive overhead for communication between the driver processes and the database server processes when there is no trigger processing work to be done. TmanTest will do the following:

```
while(total execution time of this invocation of TmanTest < THRESHOLD
            and work is left in the task queue) {
   get a task from the task queue and execute it
   yield the processor so other Informix tasks can use it (call the Informix mi_yield routine [Info98])
}
if task queue is empty
     return TASK_QUEUE_EMPTY
else
     return TASKS_REMAINING
```

The driver program will wait for T time units if the last call to TmanTest() returns TASK_QUEUE_EMPTY. Otherwise, the driver program will immediately call TmanTest() again. The default value of THRESHOLD will be 1/4 second also, to keep the task switch overhead between the driver programs and the Informix processes reasonably low, yet avoid a long user-defined routine (UDR) execution. A long execution inside TriggerMan should be avoided since it could result in higher

probability of faults such as deadlock or running out of memory. Keeping the execution time inside TriggerMan reasonably short also avoids the problem of excessive lost work if a rollback occurs during trigger processing.

Tasks can be one of the following:

1. process one token to see which rules it matches
2. run one rule action
3. process a token against a set of conditions
4. process a token to run a set of rule actions triggered by that token

Task types 1 and 2 are self-explanatory. Tasks of type 3 and 4 can be generated if conditions and potential actions (triggerID structures containing the "rest of the condition") in the predicate index are partitioned in advance so that multiple predicates can be processed in parallel. An example of when it may be beneficial to partition predicates in advance is when there are many rules with the same condition but different actions. For example, suppose there are M rules of the form:

```
create trigger T_K
from R
when R.company = "IBM"
do raise event notify_user("user K", R.company, R.sharePrice)
```

for K=1..M. If M is a large number, a speedup can be obtained by partitioning this set of triggers into N sets of equal size. This would result in a predicate index substructure like that illustrated in Figure 6.

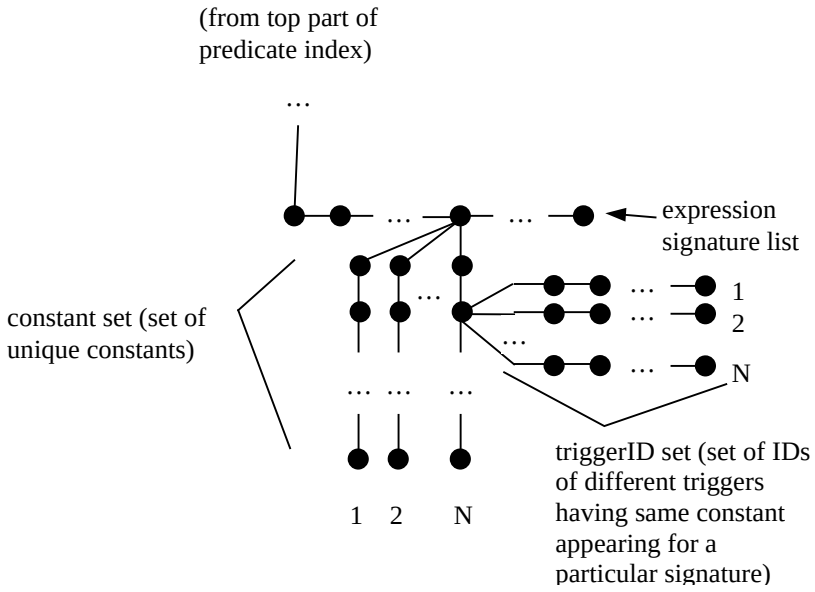


Figure 6. Illustration of partitioned constant sets and triggerID sets to facilitate concurrent processing.

Here, the triggerID set would contain references to triggers T_1 … T_M. These references would be partitioned round robin into N subsets of approximately equal size.

We adopt an approach where constant sets and triggerID sets will be converted dynamically from a non-partitioned in-memory list format to a partitioned in-memory list format when their size exceeds a certain limit. As part of future work we will determine an appropriate value for this threshold [Kony98]. When these sets get *extremely* large, they will be converted to a database table format as outlined earlier.

If no partitioned constant sets or triggerID sets are encountered when processing a token, a single thread of control completes execution of the task of processing that token. However, when a token arrives at a partitioned constant set or triggerID set, a task of the following form will be created for each partition and placed in the task queue:

<operation-code,token,pointer-to-set-to-process>

The operation-code tells what type of operation this is (process token for constant set or process token for triggerID set). The token is the update descriptor to process for the set. The pointer-to-set-to-process is the address of the appropriate sub-list in the predicate index to process for the token.

Concurrent processes can access the discrimination network. It is read during token processing, and written when triggers and trigger sets are created and destroyed. The system must be careful not to corrupt the structure when concurrent reads and writes occur. Since trigger creation is not likely to be frequent, initially we will use a simple lock on the entire discrimination network structure. This will ensure that writers operate alone in the system when they are updating the network, but allow any number of concurrent readers. Finer granularity write-locking strategies will be adopted if necessary to allow frequent trigger creation and deletion to support the needs of real applications.

At this point it is worthwhile to return to the topic of rule action concurrency. Even for situations with only one rule, it is possible that a single tuple update could cause many new tuple combinations to match the rule condition. The rule action needs to be fired for each of these combinations. It is important to be able to do the action firing in parallel for high scalability. For example, consider a situation with the following two tables representing products and their prices, and alert requests placed by users interested in when the price of a particular product goes above or below some limits:

```
product(pid, name, price)
alert(pid,personName,emailAddr,low_limit,high_limit)
```

When a product price changes, the following trigger notifies each person for which that product price is outside the specified limit for that person:

```
create trigger notify
from product, alert
when product.pid = alert.pid
and (product.price < low_limit or product.price > high_limit)
do raise event price_alert(emailAddr, product.pid, price, low_limit, high_limit)
```

If the number of alert rows is large, it is possible that a single update to one product price could cause thousands of event notifications to be sent. To improve scalability in this situation on a parallel machine, the alert table can be stored partitioned across the disks. Then, suppose an update token t is generated describing a single update to one product price. We introduce a user-defined routine raise_event that can be executed as part of a query invocation. Then when token t arrives, we can perform the above trigger as shown below. Here, t.pid and t.price are constants extracted from t. The || symbol is a user-defined operator used to pack a series of values together into one vector of values so it can be passed to raise_event as a single argument.

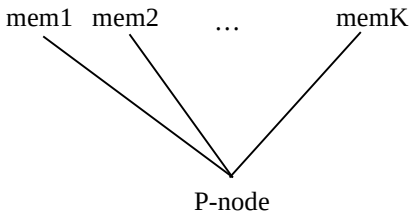
```
select count(raise_event("price_alert",emailAddr || t.pid || price || low_limit || high_limit))
from alert
where t.pid = alert.pid
and (t.price < low_limit or t.price > high_limit)
```

If alert is partitioned, then this query will run in parallel on a parallel machine. The raise_event UDR will be invoked on the result tuples in parallel. This gives scalable trigger action processing for this rule, in which a single-tuple update triggered many actions.

The concept used in the above example can be generalized as follow. Consider a trigger of the form:

```
create trigger T
from R1, …, RN
[on event]
when cond
do action
```

Such a trigger will be processed using a Gator network [Hans97,Hans97b]. For rules with join conditions, just above the P-node in the Gator network is a set of memory nodes (alpha and beta nodes) mem1… memK, as illustrated below:



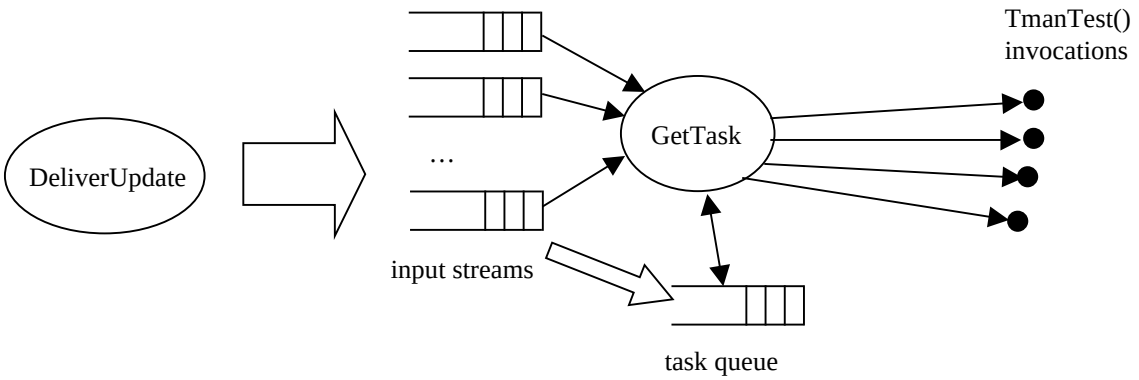
Suppose a token t arrives at the I'th memory node, memI. Then the following query will be formulated and run:

```
select count(action')
from mem1,… memI-1, memI+1, … memK
where cond'
```

Here, action' is the action of T, modified to contain values drawn from a token (or a temporary table as outlined in [Hans97]). This action' is now a procedure that contains a sequence of one or more database commands or **raise event** commands. If the query processor is designed to run queries in parallel, then the above query can execute in parallel. To get maximum speedup, large memory nodes in the set {mem1, mem2, … memK} need to be partitioned across the available disks. If the memory nodes are virtual, then the underlying tables need to be partitioned if they are large to support efficient rule processing. The database administrator must partition these underlying tables. TriggerMan will partition stored memory nodes automatically if they are large. Initially, a simple heuristic will be used to partition tables beyond a fixed size. Improvements on this heuristic are a subject for future research.

To this point we have introduced a number of different types of work that TriggerMan is required to do. To process different kinds of tasks, and get scalability for handling all kinds of tasks, we use a multi-tasking architecture built around a task queue as mentioned previously. Multiple TriggerMan driver processes can call the TmanTest() UDR concurrently to achieve task-level parallelism on an SMP machine. Each TmanTest() invocation can run in its own Informix process (Vproc) running on its own processor if a free one is available. Furthermore, data sources call the DeliverUpdate UDR to notify TriggerMan that new update descriptors have arrived. This UDR can be called either in the action of a

local Informix trigger (for a local data source) or by a data source application program (for a remote data source). A diagram of this arrangement is shown below:



Here, there are multiple input streams. A stream can represent all the tokens arriving from one connection, or all the tokens arriving from one data source. In the former case we say the connection has the *single stream* property. In the later case we say it has the *multiple stream property*. Input streams in turn have properties that describe them. One of these properties is the *serialization* property, which can be either serial or non-serial. A serial input stream must have its update descriptors processed to completion serially. Tokens from a non-serial input stream can be processed out of order. Another property of an input stream is its *recoverability* property, which can be either recoverable or non-recoverable. Properties are assigned to connections and data sources when they are defined. A complete discussion of how these properties are assigned is beyond the scope of this paper.

The GetTask bubble in the preceding diagram represents the function that TmanTest invocations call to get new tasks to perform. If the task queue is too short when GetTask is called, additional tasks will be brought over from the input streams and placed in the task queue, if new updates are available. GetTask cooperates with DeliverUpdate and monitors the task queue to make sure that (1) new updates are processed in a timely, fair manner, and (2) the serialization and recoverability properties of the data sources are observed.

From the point of view of scalability, GetTask is important because enough tasks must be available to keep all the TmanTest invocations busy to get maximum performance. For example, for a non-serial data stream with a high update rate, multiple tasks must be moved immediately onto the task queue so they are ready to be processed concurrently by different TmanTest invocations.

A mutual-exclusion semaphore guards the task queue. We believe that a single task queue with a single mutual-exclusion semaphore guarding it will not be a bottleneck. This is due to the fact that the time it takes to carry out a task, such as testing a token against the discrimination network or running a trigger action, will typically be much longer than the time required to dequeue the task. If this proves untrue for real applications, then multiple task queues could be used.

8. Trigger Application Design

The trigger system proposed in this paper is designed to be highly scalable. We feel that this architecture will make it easier for application designers to develop trigger applications since they will not need to be overly concerned that they are creating too many triggers for the system to handle efficiently. However, in some cases it is possible for trigger application designers to “code around” some situations that would require creating large number of triggers. This may be useful even if the trigger

system being used is highly scalable. For example, it may not be convenient for the developer to keep track of and manipulate large numbers of triggers.

Difficulties associated with the use of large quantities of triggers can sometimes be avoided by designing trigger application logic to use a small number of triggers, plus pattern data stored in tables, instead of a large number of triggers. As an example, an application might create one million triggers of the form:

```
create trigger t_I from R on insert to R when R.a=CONST_I do …
```

for I ranging from 1 to a million. An alternative would be to create a table called t_CONSTANTS, and a trigger t, as follows:

```
t_CONSTANTS(triggerID, const_value)

create trigger t
from R, t_CONSTANTS
on insert to R
when R.a=t_CONSTANTS.const_value
do …
```

Here, a single two-tuple-variable trigger has replaced a large number of single-tuple-variable triggers. When an insert to R, represented as a token t, occurs, the system would run the following query:

```
select * from t_CONSTANTS where t.a=t_CONSTANTS.const_value
```

Here, t.a is a constant extracted from t. The results of this query would then be passed to the trigger action. This example illustrates that there are a variety of choices for designing a trigger application. Even if the trigger system in use is scalable, like the one proposed here, application designers should be aware of alternative implementations and their strengths and weaknesses.

9. Related Work

There has been a large body of work on active database systems, but little of it has focussed on predicate indexing or scalability. Representative works include HiPAC, Ariel, the POSTGRES rule system, the Starburst Rule System, A-RDL, Chimera, and Ode [Wido96]. Work by Hanson and Johnson focuses on indexing of range predicates using the interval skip-list data structure [Hans96b], but this approach does not scale to very large numbers of rules since it may use a large amount of main memory. Work on the Rete [Forg82] and TREAT [Mira87] algorithms for efficient implementation of AI production systems is related to the work presented here, but the implicit assumption in AI rule system architectures is that the number of rules is small enough to fit in main memory. Additional work has been done in the AI community on parallel processing of production rule systems [Acha92], but this does not fully address the issue of scaling to large numbers of rules. Work by Hellerstein on performing selections after joins in query processing [Hell98] is related to the issue of performing expensive selections after joins in Gator networks.

10. Conclusion

This paper describes the architecture of a truly scalable trigger system. As of the date of this writing, this architecture is being implemented as an Informix DataBlade along with a console program, a driver program, and data source programs. The architecture presented is a dramatic advance over what is currently available in database products. It also generalizes earlier research results on predicate indexing and improves upon their limited scalability [Forg82,Mira87,Hans90,Hans96]. This architecture could be

implemented in any object-relational DBMS that supports the ability to execute SQL statements inside user-defined routines. A variation of this architecture could also be made to work as an external application, communicating with the database via a standard interface (ODBC).

One topic for future research includes developing ways to handle temporal trigger processing [Hans97,AlFa98] in a scalable way, so that large numbers of triggers with temporal conditions can be processed efficiently. Another is to develop ways to do parallel searching of the in-memory list and index organizations to improve throughput for small and medium numbers of triggers. The current architecture already takes advantage of parallelism for large numbers of predicates constant values organized in a table since the SQL queries posed against the table will can run in parallel on a parallel machine. Finally, a third potential research topic is to develop a technique to make the implementation of the main-memory and disk-based structures used to organize the constant sets illustrated in Figure 5 extensible, so they will work effectively with new operators and data types. In the end, the results of this paper and the additional research outlined here can make highly efficient, scalable, and extensible trigger processing a reality.

Bibliography

[Acha92] Acharya, A., M. Tambe, and A. Gupta, "Implementation of Production Systems on Message-Passing Computers," IEEE Transactions on Knowledge and Data Engineering, 3(4), July 1992.

[AlFa98] Al-Fayoumi, Nabeel, *Temporal Trigger Processing in the TriggerMan Active DBMS*, Ph.D. dissertation, Univ. of Florida, 1998. In preparation.

[Date93] Date, C. J. And Hugh Darwen, *A Guide to the SQL Standard*, 3rd Edition, Addison Wesley, 1993.

[Forg82] Forgy, C. L., Rete: "A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," *Artificial Intelligence*, vol. 19, pp. 17-37, 1982.

[Hans90] Hanson, Eric N., M. Chaabouni*, C. Kim and Y. Wang*, "A Predicate Matching Algorithm for Database Rule Systems," *Proceedings of the ACM-SIGMOD Conference on Management of Data, pp. 27*1-280, Atlantic City, NJ, June 1990.

[Hans96] Hanson, Eric N., "The Design and Implementation of the Ariel Active Database Rule System," *IEEE Transactions on. Knowledge and Data Engineering*, vol. 8, no. 1, pp. 157-172, February 1996.

[Hans96b] Hanson, Eric N. and Theodore Johnson, "Selection Predicate Indexing for Active Databases Using Interval Skip Lists," *Information Systems*, vol. 21, no. 3, pp. 269-298, 1996.

[Hans97] Hanson, Eric N., N. Al-Fayoumi, C. Carnes, M. Kandil, H. Liu, M. Lu, J.B. Park, A. Vernon, "TriggerMan: An Asynchronous Trigger Processor as an Extension to an Object-Relational DBMS," University of Florida CISE Dept. Tech. Report 97-024, December 1997. http://www.cise.ufl.edu.

[Hans97b] Hanson, Eric N., Sreenath Bodagala, and Ullas Chadaga, "Optimized Trigger Condition Testing in Ariel using Gator Networks," University of Florida CISE Dept. Tech. Report 97-021, November 1997. http://www.cise.ufl.edu.

[Hans98] Hanson, Eric N., I.C. Chen, R. Dastur, K. Engel, V. Ramaswamy, W. Tan, C. Xu, "A Flexible and Recoverable Client/Server Database Event Notification System," *VLDB Journal*, vol. 7, 1998, pp. 12-24.

[Hell98] Hellerstein, J., "Optimization Techniques for Queries with Expensive Methods**,"** to appear, *ACM Transactions on Database Systems (TODS)*, 1998. Available at

www.cs.berkeley.edu/~jmh.

[Info98] "Informix Dynamic Server, Universal Data Option," http://www.informix.com.

[Kand98] Kandil, Mohktar, *Predicate Placement in Active Database Discrimination Networks*, PhD Dissertation, CISE Department, Univ. of Florida, Gainesville, 1998. In preparation.

[Kony98] Konyala, Mohan K., *Predicate Indexing in TriggerMan,* MS thesis, CISE Department, Univ. of Florida, Gainesville, 1998. In preparation.

[Mira87] Miranker, Daniel P., "TREAT A Better Match Algorithm for AI Production Systems," *Proceedings of the AAAI Conference,* August 1987, pp. 42-47.

[Wido96] Widom, J. And S. Ceri, *Active Database Systems*, Morgan Kaufmann, 1996.